AD Master Class: Advanced Adjoint Techniques

Monte Carlo

Viktor Mosenkis
`viktor.mosenkis@nag.co.uk`

12 November 2020

Experts in numerical algorithms
and HPC services

AD Masterclass Schedule and Remarks

AD Masterclass Schedule

- 1 October 2020 | Checkpointing and external functions 1
- 15 October 2020 | Checkpointing and external functions 2
- 29 October 2020 | Guest lecture by Prof Uwe Naumann on Advanced AD topics in Machine Learning
- 12 November 2020 | Monte Carlo
- 19 November 2020 | Guest lecture by Prof Uwe Naumann on Adjoint Code Design Patterns applied to Monte Carlo
- 25 November 2020 |Computing Hessians

Remarks

- Please submit your questions via the questions panel at any time during this session, these will be addressed at the end.
- A recording of this session, along with the slides will be shared with you in a day or two.

Dialogue

We want this webinar series to be interactive (even though it's hard to do)

- We want your feedback, we want to adapt material to your feedback
- Please feel free to contact us via email to ask questions at any time
- We'd love to reach out offline, discuss what's working, what to spend more time on
- For some orgs, may make sense for us to do a few bespoke sessions

- This is an advanced course
- We assume that you are familiar with the material from the first Masterclass series
- You will get access to the materials from the first Masterclass series via email in a day or two
- Also it is not a pre-requisite we recommend to review the material from the previous series
- We will try to give references to the previous Masterclass series whenever possible

Outcomes

- Learn how to write efficient adjoint code for Monte Carlo based simulations, on a small example code. We will touch on
  - Different strategies to reduce memory requirements
  - Parallelization of
    - primal and
    - adjoint
- Discuss the results on a more realistic Monte Carlo code (from finance).

What do we mean by Monte Carlo (MC) based code?

Code that contains mutually independent loop iterations. E.g.

```
template <typename T>
T f(const std::vector<T>& x, const double& r) {...}

int main() {
  initialize(x); generate_random_numbers(r);
  sum = 0.0;
  for (int i = 0; i < num_mcpath; i++) {
    sum += f(x, r[i]); // mutually independent
  }
  res = sum / num_mcpath;
  res = pow(res, 2);
  return 0;
}
```

Linear dependency in `sum` is allowed

Parallelization of primal MC code with OpenMP

```cpp
template <typename T>
T f(const std::vector<T>& x, const double& r) {...}

int main() {
  initialize(x); generate_random_numbers(r);
  sum = 0.0;
#pragma omp parallel for reduction(+:sum)
  for (int i = 0; i < num_mcpath; i++) {
    sum += f(x, r[i]);
  }
  res = sum / num_mcpath;
  res = pow(res, 2);
  return 0;
}
```

Concurrent writes in `sum` handled by reduction.

Challenges for Adjoint code with MC

Efficient adjoint implementation of MC code must address the following problems

- High number of loop iteration (paths) leads to high memory requirements. Although each loop iteration is typically small the overall memory usage can be high. E.g. one path requires 100KB of tape.
  - ☐ 1k paths $\approx$ 100MB
  - ☐ 10k paths $\approx$ 1GB
  - ☐ 100k paths $\approx$ 10GB
- Parallelization of the tape interpretation

Strategies to reduce memory usage

Checkpoint each Monte Carlo path

```
int main() {
  ...
  for (int i = 0; i < n; i++)
    tape->register_variable(x[i])

  auto p0 = DCO_M::global_tape->get_position();
  //MC code
  for (int i = 0; i < num_mcpath; i++)
    sum += f_make_gap(x, r[i]); //create a gap in the tape

  res = sum / num_mcpath;
  res = pow(res, 2);
  ...
  dco::derivative(res) = 1.0;
  tape->interpret_adjoint_and_reset_to(p0);
}
```

Checkpoint each Monte Carlo path: Make gap

```
T f_make_gap(std::vector<T>& x, const double& r) {
  auto D=tape->template create_callback_object<DCO_EAO_T>()
  T y;
  std::vector<double> xp(x); //copy inputs to passive

  y = f(xp, r[path_number]); //compute value with double

  DCO_M::global_tape->register_variable(y);
  //write checkpoint
  D->write_data(x);
  D->write_data(r);
  D->write_data(y);
  tape->insert_callback(f_fill_gap, D);
  return y;
}
```

Checkpoint each Monte Carlo path: Fill gap

```
void f_fill_gap(DCO_M::external_adjoint_object_t* D) {
  auto p0 = DCO_M::global_tape->get_position();
  //restore x, r and y
  auto const &x = D->read_data<std::vector<DCO_T>>();
  auto const &r = D->read_data<double>();
  auto const &y = D->read_data<DCO_T>();

  DCO_T y_a = f(x, r); // record the tape of the path

  dco::derivative(y_a) = dco::derivative(y);
  //compute the adjoint and free the tape
  tape->interpret_adjoint_and_reset_to(p0);
}
```

Checkpoint each MC path: Remarks

- Advantages
  - Universal approach works for not mutually independent loops
- Disadvantages
  - Tape size grows with the number of MC paths
  - MC paths are computed twice, once to create the gap (without taping) and once to fill it with taping.
  - Checkpoint callbacks can decrease performance
- Implementation tips
  - Checkpoint paths in chunks (batches) can improve
    - performance and
    - required tape size
  - Reduce size of the checkpoint data by sharing information (exploit mutual independence of loop iterations)

Early pathwise interpretation

Interpret MC path directly after it has been recorded.

```
int main() {
  ..
  for (int i = 0; i < n; i++)
    tape->register_variable(x[i]);
  auto p0 = tape->get_position();
  //MC code
  for (int i = 0; i < num_mcpath; i++) {
    sum += f(x, r[i]);
    //requires knowledge of adjoint of sum
    dco::derivative(sum) = 1.0 / num_mcpath;
    tape->interpret_adjoint_and_reset_to(p0);
    sum = dco::value(sum);
  }
  res = sum / num_mcpath;
}
```

Early pathwise interpretation: Remarks

- Advantages
  - Tape size independent from the number of MC paths
  - MC paths are computed only once
  - No checkpointing required
  - Potentially better usage of cache due to small tape size compared to naive approach
- Disadvantages
  - Requires the knowledge of the adjoint of `sum`
- Implementation tips
  - Interpret paths in chunks can improve performance for very small paths

Pathwise interpretation

First gap the MC simulation and compute the adjoint of the output of MC, fill the gap using early pathwise interpretation

```
int main() {
  ..
  for (int i = 0; i < n; i++)
    tape->register_variable(x[i])
  auto p0 = DCO_M::global_tape->get_position();

  //MC code
  g_make_gap(x,r,sum)

  res = sum / num_mcpath;
  res = pow(res, 2);

  dco::derivative(res) = 1.0;
  tape->interpret_adjoint_and_reset_to(p0);
}
```

Pathwise interpretation: Make Gap

```
template <typename T>
void g_make_gap(const std::vector<T>& x, const std::
    vector<double>& r, T& sum) {
  auto D = tape->create_callback_object<DCO_EAO_T>();
  for (int i = 0; i < x.size(); i++)
    xp[i] = dco::value(x[i])
  // run MC without taping
  for (int i = 0; i < num_mcpath; i++)
    sum += f(xp, r[i]);

  DCO_M::global_tape->register_variable(sum);
  D->write_data(x);
  D->write_data(r);
  D->write_data(sum);
  tape->insert_callback(g_fill_gap, D);
}
```

Pathwise interpretation: Fill Gap

```
void g_fill_gap(DCO_M::external_adjoint_object_t* D){
  auto p0 = DCO_M::global_tape->get_position();

  //restore data from checkpoint x, r, sum
  double sum_a = dco::derivative(sum)

  for (size_t i = 0; i < num_mcpath; i++) {
    sum += f(x, r[i], y);
    dco::derivative(sum) = sum_a; //adjoint of MC output

    DCO_M::global_tape->interpret_adjoint_and_reset_to(p0);
  }
}
```

Pathwise interpretation: Remarks

- Advantages
  - Tape size independent from the number of MC paths
  - the adjoint of `sum` is computed automatically
  - only one checkpoint required
  - Potentially better usage of cache due to small tape size compared to naive approach
- Disadvantages
  - MC paths are computed twice, once to create the gap (without taping) and once to fill it with taping
- Implementation tips
  - Interpret paths in chunks can improve performance for very small paths

Strategy for parallelization

Parallelization of the tape interpretation

Basic idea

- Primal
  - compute MC paths in parallel
  - concurrent write while updating `sum`
- Adjoint
  - interpret MC paths in parallel
  - concurrent write for updating input `x`

Parallelization of pathwise interpretation

- in make_gap the MC simulation can be parallelized in the same way as normal MC code
- in fill_gap
  - each thread creates its own tape
  - each thread gets a local copy of input `x`
  - each paths is interpreted directly after recording
  - the partial adjoints of input `x` are accumulated in the local copy of inputs
  - after the MC simulation is done the partial adjoints stored in the local copy's can be gathered in `x` (concurrent write).

Parallelization of early pathwise interpretation

Apply the same steps as in the fill_gap routine of pathwise interpretation

- each thread creates its own tape
- each thread gets a local copy of input `x`
- each paths is interpreted directly after recording
- the partial adjoints of input `x` are accumulated in the local copy of inputs
- after the MC simulation is done the partial adjoints stored in the local copy's can be gathered in `x` (concurrent write).

Call option on a basket example

We developed an in house code that computes a call option on a basket using MC simulation

- driven by multi factor local volatility model
- local volatility surfaces are gridded into a lookup table
- code is structured for vectorization over the MC paths
- code is parallelized over the MC paths with OpenMP (as outlined before)
- spline interpolation, BLAS and LAPACK routines from the NAG AD Library are used
- Adjoint Code Design Pattern (ACDP) are applied to that code implementing some of the strategies for MC codes discussed today

Call option on a basket example

Memory usage

Num paths	before MC	Overall		
		plain	pathw	early
10k	0.22GB	2.64GB	0.33GB	0.27GB
100k	2.08GB	26.3GB	3.1GB	2.5GB

Call option on a basket example

Parallelization scalability with 100k MC paths

	Number of Threads			
	1	**4**	**12**	**24**
primal	6.1s (1)	1.58s (3.9)	0.83s (7.3)	0.52s (11.7)
plain	9.9 (60.7s)	9.4 (14.9s)	6.2 (5.18s)	9.7 (5.03s)
pathw	12 (73.1s)	12.7 (20.2s)	9.7 (8.01s)	13 (6,74s)
early	9 (54.9s)	8.8 (13.9s)	6 (4.99s)	8.3 (4.32s)

Summary

In this Masterclass we

- learned different ways to compute adjoints of MC based code without running out of memory and discussed their advantages and disadvantages
  - checkpoint each MC path
  - early pathwise interpretation
  - pathwise interpretation
- discussed how to efficiently parallelize tape interpretation in MC based codes

AD Master Class 5: On Adjoint Code Design pattern

In the next class our guest lecturer Prof. Uwe Naumann will discuss how to exploit common patterns shared around many simulation codes to reduce the effort required to create efficient adjoints. This talk covers patterns for

- Monte Carlo adjoints
- implicit function theorem
- checkpointing

You will see a survey on your screen after exiting from this session.

We would appreciate your feedback.

We are now moving on the Q&A Session